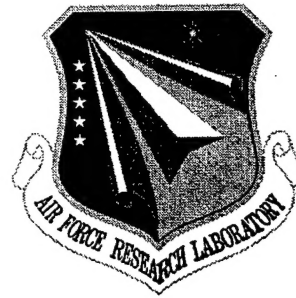


AFRL-IF-RS-TR-1998-236
Final Technical Report
January 1999



A FRAMEWORK FOR THE CERTIFICATION AND EVALUATION OF REAL-TIME SAFETY-CRITICAL INTELLIGENT SYSTEMS

University of Illinois at Chicago

Jeffrey J.P. Tsai

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

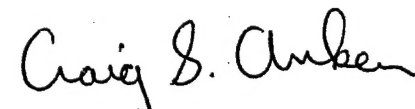
**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

1 9990217036

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1998-236 has been reviewed and is approved for publication.

APPROVED:



CRAIG S. ANKEN
Project Engineer

FOR THE DIRECTOR:



NORTHROP FOWLER, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE January 1999		3. REPORT TYPE AND DATES COVERED Final Jul 95 - Oct 97
4. TITLE AND SUBTITLE A FRAMEWORK FOR THE CERTIFICATION AND EVALUATION OF REAL-TIME SAFETY-CRITICAL INTELLIGENT SYSTEMS			5. FUNDING NUMBERS C - F30602-95-1-0035 PE - 625581 PR - 5581 TA - 27 WU - 95	
6. AUTHOR(S) Jeffrey J.P. Tsai				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Illinois at Chicago Department of Electrical and Computer Science 851 South Morgan Street Chicago IL 60607			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTB 525 Brooks Road Rome NY 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-1998-236	
11. SUPPLEMENTARY NOTES Air Force Research Laboratory Project Engineer: Craig S. Anken/IFTB/(315) 330-4833				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The concept of software architecture has recently emerged as a new way to improve our ability to effectively construct large scale software systems. However, there is no formal architecture specification language available to model and analyze temporal properties of complex real-time systems. In this paper, an object-oriented logic-based architecture specification language for real-time systems is discussed. Representation of the temporal properties and timing constraints, and their integration with the language to model real-time concurrent systems is given. Architecture based specification languages enable the construction of large system architectures and provide a means of testing and validation. In general, checking the timing constraints of real-time system is done by applying model checking to the constraint expressed as a formula in temporal logic. The complexity of such a formal method depends on the size of the representation of the system. It is possible that this size could increase exponentially when the system consists of several concurrently executing real-time processes. This means that the complexity of the algorithm will be exponential in the number of processes of the system and thus the size of the system becomes a limiting factor. Such a problem has been defined in the literature as the "state explosion problem". This paper proposes a method of incremental verification of architectural specifications for real-time systems. The method has a lower complexity in a sense that it does not work on the whole state space, but only on a subset of it that is relevant to the property to be verified.				
14. SUBJECT TERMS Real-Time AI, Performance Evaluation, Monitoring, Anytime Algorithms			15. NUMBER OF PAGES 28	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

THIS QUALITY INSPECTED 1

Abstract

The concept of software architecture has recently emerged as a new way to improve our ability to effectively construct large scale software systems. However, there is no formal architecture specification language available to model and analyze temporal properties of complex real-time systems. In this paper, an object-oriented logic-based architecture specification language for real-time systems is discussed. Representation of the temporal properties and timing constraints, and their integration with the language to model real-time concurrent systems is given. Architecture based specification languages enable the construction of large system architectures and provide a means of testing and validation. In general, checking the timing constraints of real-time systems is done by applying model checking to the constraint expressed as a formula in temporal logic. The complexity of such a formal method depends on the size of the representation of the system. It is possible that this size could increase exponentially when the system consists of several concurrently executing real-time processes. This means that the complexity of the algorithm will be exponential in the number of processes of the system and thus the size of the system becomes a limiting factor. Such a problem has been defined in literature as the "state explosion problem".

We propose a method of incremental verification of architectural specifications for real-time systems. The method has a lower complexity in a sense that it does not work on the whole state space, but only on a subset of it that is relevant to the property to be verified.

Keywords: Model-checking, architecture, Requirements specification, labeled transition system.

1 Introduction

For the construction of large concurrent and real-time systems, it is useful to build an architecture level description of the system. This enables representation of how the components relate to each other in a global way in addition to providing a means of constructing and reusing them. This will also help to execute and check the performance of the system before it is actually built. But, for this to be feasible, the architecture must be specified in a way that allows easy analysis and checking of the runtime constraints of the specification. The execution of such architecture descriptions can be applied in stages of a system development, for early prototyping to check consistency, for verification of various aspects of concurrency and timing, for testing that communication between components satisfies architectural constraints, etc.

One of the most important issues concerning the real-time systems is to verify that the system meets its timing constraints. Majority of such systems contain multiple processes executing in parallel. For representing such systems, one needs an architecture level language supporting object-oriented features, a way of representing data, knowledge and operations, communication between objects; and a semantic foundation using which the properties of the system can be verified.

The architecture specification language used here is an object oriented specification language based on logic [1]. The syntax consists of a set of object and activity frames. The object hierarchies are represented through inheritance relations and activities are modeled as processes. The semantics is defined using an extended Horn clause logic. Standard Horn-clause logic provides neither a hierarchical structure with property inheritance nor an exception mechanism to its rules and constraints. We use a variant of Horn-clause logic supplemented with these two mechanisms via the concept of non-monotonic reasoning to establish a formal foundation for the object-oriented architecture specification language. The non-monotonicity is a desired property that the semantics must support because in the hierarchy, for example, some properties of objects may be overridden during the course

of software development. The language supports most of the mechanisms for modeling concurrent distributed systems such as AND and OR-parallelism, non-determinism, synchronous and asynchronous communication.

The correctness of the functionality of the system is determined by computing the models (non-monotonic extensions) of the horn clause logic program. The time-dependent aspects are modeled using a temporal logic framework. The correctness of the specification with respect to a wide variety of temporal properties is determined using model checking [2,3,4,5,6]. Temporal logic has been used extensively for the specification and verification of concurrent systems. It was found especially useful in proving properties of concurrent programs describing systems at any level of abstraction, and for compositional reasoning. Different systems of temporal logic use different modalities and notations. However, they generally rely on either linear or branching time logics. In a linear time logic, the temporal modalities are defined with respect to a single path which the program follows. Typical linear time operators include "always," "sometimes," "next," and "until." Properties expressed by branching time logics include "inevitably," (for all futures, sometime) "potentially," (for some future, sometime) and "invariably" (for all futures, always). With temporal logic based frameworks, it is difficult to specify notions of absolute time. Generally, only relative orderings between processes can be stated.

2 Background and Significance

In the following, we use the syntax and semantics of an extension of modal mu calculus (called $RT\mu$) [7,8]. [7] gives an algorithm for model-checking using this extension of mu-calculus and a labeled transition system as a representation of the system. This algorithm has a small polynomial time complexity. Temporal aspects of the architectural specification are also discussed and a method of verifying its timing constraints through model checking is given.

Several methods for model checking proposed so far in the literature basically focus on deriving a reduced, compact representation of state-spaces before verifying the state-space against a property [9,11]. Nevertheless, the following problem remains: whenever a change is induced in the specification, we have to rebuild the state-space again from scratch, however minor that change might be. *Incremental* model checking is an efficient state-space exploration based method for the verification of systems that frequently undergo changes in their design and requirement phases. Instead of building the representation of the system from scratch after any change is

introduced into the system, the representation of the system is *incrementally* modified so that the new state-space satisfies the revised specification. This gives another use of this method: if the change is local, i.e, applies only to a component of the system, then one can analyze the impact of that change on other components and on the overall behavior of the system. [5] gives an incremental model checking algorithm for the alternation free fragment of modal mu-calculus for a system represented by a labeled transition diagram. Their algorithm takes time linear in the size of the labeled transition diagram in the worst case, but in the best case its complexity is linear in the magnitude of the change applied.

In terms of applicability of the algorithm to practical problems, our method can be applied for efficient verification of temporal properties of systems that frequently experience changes in their requirements, which demands changes in their design i.e, the state space representation. However, in some cases, the time complexity of such an incremental algorithm could be as bad as the standard model-checking algorithm [7]. Such cases arise when the change made in the state-space propagates through the whole state-space.

3 Motivation

Verification of safety critical timing constraints of a specification representing a real-time concurrent system can be a very time-consuming process. The complexity of model checking for a given specification, represented as a model for some sentence in temporal logic, depends on the number of states of the model. Moreover, the size of the state space increases exponentially with the number of concurrent processes in the system. The complexity of the temporal logic formula is another factor determining the performance of the model checking. Formally, the complexity of the algorithm [3] is $O((|M||\phi|)^c)$ where $|M|$ is the size of the model M corresponding to the specification, ϕ is the sentence in the temporal logic and c is a constant that depends on the complexity of the formula ϕ . In practice, all the formulas needed to express the timing properties of real-time systems have a complexity of not more than $c = 2$. Hence, the main issue is to avoid the state explosion since size of M depends on the number of states. One of the potential solutions is to build an abstraction of the system that has a smaller state space and yet the same behavior (as suggested in [10]) and then apply the model checking to the abstraction. Another solution is to perform incremental runs of the algorithm on the system. This will avoid repeated reference to the whole state space of the system and hence the performance of each incremental run will

not depend on the size of the whole state space. The incremental algorithm in the best case depends only on the size of the change made to the model (transition system). [5] gives an example to which incremental model checking is applied to verify certain timing properties and shows that every incremental run of the algorithm can be performed in constant time irrespective of the number of processes of the system.

4 Modal Mu-calculus

Modal mu-calculus is a powerful temporal logic that can express the safety, liveness and fairness properties of real-time systems. It is shown in [3] that a restricted fragment of propositional modal mu-calculus is adequate for formalizing most temporal reasoning about distributed real-time programs and they also give a small polynomial time complexity model checking algorithm for any specification that can express all temporal assertions found in practice.

The performance of the model checking algorithm depends on the complexity of the fix-point assertion to be verified. For a formula in which the alternating depth [3] of least and greatest fix-points is one, the algorithm runs in linear time. Fortunately, almost every temporal property of any real-time system in practice can be expressed by such a formula. Syntax and semantics of $RT\mu$ is explained in [7].

5 Temporal aspects of the architectural specification

[7] gives constructs to represent timing constraints of specifications. In particular it gives constructs for representing clock activities and shows how these are implemented by integrating them with the underlying parallel logic program. Two kinds of timing activities can be modeled here - the periodic and the sporadic processes. The language also uses certain built in temporal operators like *next*, *henceforth*, *eventually*, *until*, and *precede*. These constructs have a straight forward conversion into $RT\mu$.

6 Verification of timing constraints of architectures for real time systems through Model checking

To verify a program through model checking treat the specification as a structure. Then determine whether this structure is a model for sentence of $RT\mu$ that expresses a desired property of the specification. The procedure for determining whether a structure is a model of $RT\mu$ has acceptable time-complexity. Model checking gives

us a powerful mechanism to determine the correctness of our specification relative to a wide variety of temporal properties. The properties that we want to verify are the safety properties like state invariants, global invariants, partial correctness, mutual exclusion and deadlock freedom, and liveness and fairness properties.

6.1 Expressing specifications as models

To verify a program through model checking we treat the specification as a transition system. Then we determine whether this transition system is a model for the $RT\mu$ sentences that express the desired property of the specification. A specification can be viewed as a transition system in the following way: Given a set of states of the computation, the specification tells us for each state, which other states can be reached by a single step of computation. Formally, a transition system is a triple $A = \langle \Sigma_A, S_A, R_A \rangle$, where Σ_A is the alphabet of the transition system, S_A is a set of states, and R_A is a function from $S_A \times \Sigma_A$ to 2^{S_A} . Each transition has an associated label from Σ_A . To convert a specification into the corresponding transition system, create a new state for each unique action (or alternative action) and precondition in the specification. Let Σ_A be the set of actions, preconditions and alternative actions of the specification. Then, for every two states corresponding to two actions or preconditions connected by a conjunction, create a transition between these two states labelled by the first action/precondition. Here, we can collapse any sequence of states that donot contribute to the time-dependent behaviour of the specification.

When several processes execute concurrently, each state of the execution is a combination of the states of the individual processes. To form a transition system representing the concurrent execution of the component transition systems, we form the product of the component transition system. A transition in the combined system is then due to a transition in one of the component transition systems. We add, however, a constraint on transitions in the combined system. There are situations requiring two actions of component systems to be executed simultaneously. If this constraint is met by the product of two transition systems, we call the resulting system synchronized.

Definition 1 *Given the transition systems $A_1 = \langle \Sigma_1, S_1, R_1 \rangle, \dots, A_n = \langle \Sigma_n, S_n, R_n \rangle$, the synchronized product of A_1, \dots, A_n is the transition system $A = \langle \Sigma, S, R \rangle$, where*

$$(i) \Sigma = \bigcup_{i=1}^n \Sigma_i$$

$$(ii) S = S_1 \times \dots \times S_n$$

$$(iii) \prod_{k=1}^n v_k \in R(\prod_{k=1}^n u_k, \alpha), \text{ for any } 1 \leq i, j \leq n, \text{ and } i \neq j, \text{ either}$$

- $\alpha \in \Sigma_i \cap \Sigma_j$ and $v_i \in R_i(u_i, \alpha) \wedge v_j \in R_j(u_j, \alpha)$
- $\alpha \in \Sigma_i - \Sigma_j$ and $v_i \in R_i(u_i, \alpha) \wedge v_j = u_j$.

(let $\prod_{k=1}^n$ denote the formation of an n -tuple).

Note that the synchronized product of transition systems is both associative and commutative. We could also express it as an operation over two transition systems and obtain the product of several transition systems through repeated product formation.

From the synchronized product of the processes under consideration, we construct the model intended to provide a model for the sentences of the temporal calculus to be tested.

Definition 2 *Given a transition system $A = \langle \Sigma_A, S_A, R_A \rangle$, the model $M = \langle S, R, L \rangle$ corresponding to A is*

- (i) $S = S_A$
- (ii) R is a function from S to 2^S , such that $t \in R(s)$ if for some label $\alpha \in \Sigma_A$, $t \in R_A(s, \alpha)$
- (iii) L is a function from S to 2^{Σ_A} , such that if there exist states $s, t \in S_A$ and a label $\alpha \in \Sigma_A$, and $t \in R_A(s, \alpha)$, then $\alpha \in L(t)$.

Note that the set of propositional constants of the model is the set of labels of the corresponding transition system.

7 Incremental Verification

A concurrent real-time system can be considered as built up by undergoing a series of incremental updates to its specification for example by addition or deletion of states/transitions to the transition graph representing the specification. Suppose the system satisfies a set of temporal properties. These properties need to be verified through model-checking every time the system undergoes a change. But it would be very costly to repeatedly apply model checking to the whole state space every time a transition is added/deleted from the graph. Given a

set of changes to the specification one wants to derive the new assignment of variables/properties to states and at the same time no global information should be maintained between updates. Here we consider the verification of least fix-point formulas using incremental model checking. For greatest fix-point formulas, the method is completely dual. Also, every greatest fix-point formula can be transformed into a least fix-point formula by introducing negations. In the next subsection, we first show how non-incremental model-checking can be done using the structures defined.

7.1 Preliminaries

We re-write a fix-point formula as a set E of equations of the form $X_i = f_i$ where $X_i \in Var$, the countably infinite set of variables, and f_i is a propositional constant or obtained by applying exactly one operator to variables. For example, consider the following least fix-point formula

$$F = \mu z. Q \vee (P \wedge \forall \bigcirc z).$$

This is re-written as

$$X_1 = X_4 \vee X_2$$

$$X_2 = X_5 \wedge X_3$$

$$X_3 = \forall \bigcirc z$$

$$X_4 = Q$$

$$X_5 = P$$

Let $\text{Subf}(F)$ represent the set of all sub-formulas of the fix-point formula F . Hence, for the above example, $\text{Subf}(F) = \{X_1, X_2, X_3, X_4, X_5\}$. In the following procedure, we make use of a state-variable graph SV which is defined for a formula F and a model (transition system) $M = (S, A, R)$ where S is the set of states, A is the set of actions and R is the transition relation, as follows:

$SVG = (N', E', E'')$ where $N' = \{ \langle s, X \rangle \mid s \in S, X \in \text{Subf}(F) \}$.

(i) For all states $s \in S$, $\langle s, X_i \rangle \rightarrow \langle s, X_j \rangle \in E'$ and $\langle s, X_k \rangle \rightarrow \langle s, X_j \rangle \in E'$ if $X_j = X_i \vee X_k$ or $X_j = X_i \wedge X_k$.

(ii) If $s \rightarrow s' \in R$ and $X_i = \forall \bigcirc X_j$ or $X_i = \exists \bigcirc X_j$ then $\langle s', X_j \rangle \rightarrow \langle s, X_i \rangle \in E'$.

(iii) For all states $s \in S$, $\langle s, X \rangle \rightarrow \langle s, z \rangle \in E''$ where X is the top-level fix-point formula or sub-formula (in case of nested fixpoint formulas) and z is the variable that comes under the scope of the fix-point operator of the formula X .

Definition 7.1 A variable assignment is an assignment of variables to states of the model such that a variable true in a state is assigned to that state. Formally, there is a set $v(S)$ for each state S of the model such that $v(S) = \{X | f(X, S) = \text{true}\}$, where $f(X, S)$ is true iff the variable X is true in the state S .

We associate a truth value with every node. Initially, let all the nodes be false, indicating that all variables defined in E are false in all states. Note that since we are computing least fix-point, the variable assignment for each state at any time should be lower than the variable assignment corresponding to the fix-point in the lattice of the variable assignments. This is because the method works by monotonically raising the variable assignment till it reaches fix-point. Now, we make some nodes of SV graph true so that the variable assignment reaches the least fix-point. If $\{X = P\} \in E$, where P is a propositional constant, then for all states s belonging to $V(P)$ make $\langle s, X \rangle$ true. Also, for all nodes $\langle s, X_i \rangle$ such that s has no successors and $X_i = \forall \bigcirc X_j \in E$ ($X_i = \exists \bigcirc X_j \in E$), make $\langle s, X_i \rangle$ true (false).

After the above initialization step, we start the fix-point computation. Intuitively, this computation can be considered as a repeated bottom-up evaluation of a tree T in which the nodes are labeled as $\langle X, OP \rangle$ where X is defined in E or X is a free variable occurring in the formula, and $OP \in \{\wedge, \vee, \neg, \bigcirc, -\}$ or OP is a propositional constant. We also associate a value for each node which is the set of states satisfying the variable corresponding to the node. For instance, in the above example, T would be

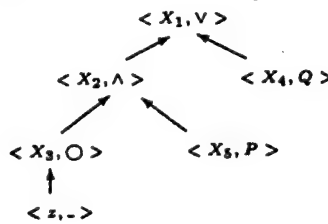


Figure 1. Formula Graph.

Every leaf of the form $\langle X, P \rangle$ where $P \in Prop$ is given the value $V(P)$ i.e, set of states in which P is true.

The leaf corresponding to the free variable z is assigned the value ϕ (since we are evaluating the least fix-point). The values of all other nodes is undefined. Each inner node $\langle X, OP \rangle$ is evaluated by applying the operator OP to its successor nodes with \wedge meaning set intersection, \vee meaning set union and \neg meaning set complement. The algorithm for evaluating the least fix-point is as follows

```

procedure evaluate(T);
begin
    for all nodes of the form  $\langle z, \_ \rangle$  do
        val( $\langle z, \_ \rangle$ ) = null;
    for all nodes of the form  $\langle X, P \rangle$  do
        val( $\langle X, P \rangle$ ) = V(P);
    repeat
        prev = val( $\langle z, \_ \rangle$ );
        Bottom-up evaluation;
        val( $\langle z, \_ \rangle$ ) = val(root);
    until prev = val(root);
end

```

val(root) at the end of the procedure is the set of states satisfying the least fix-point formula F . We now use the values of the nodes in T to update the truth values of the nodes in SV graph. That is, for each node $\langle X, OP \rangle$ in T and for each state $s \in \text{val}(\langle X, OP \rangle)$ make all nodes in SV graph of the form $\langle s, X \rangle$ true if they are already not true. This gives us the new variable assignment corresponding to the fix-point formula F .

7.2 Incremental Algorithm

This algorithm takes as input a set δ of changes to the transition system, the set E of equations corresponding to the least fix-point formula F to be verified and the state-variable graph SV and produces as output a new graph SV_1 corresponding to the new variable assignment. Note that the input SV graph should be the one obtained by an application of the non-incremental algorithm to F and the model before the changes δ are introduced into the transition system. The changes may be addition or deletion of transitions, or changes in the set of propositions

that are true in a state in the labeled transition system (LTS). In the following, $\forall \bigcirc$ and $\exists \bigcirc$ represent universally and existentially quantified next time operators respectively. The algorithm runs in three phases. In the first phase (*phases 0 and 1*) we compute the direct effects of the changes on the state-variable graph. The second phase involves updating values of SV graph in order to account for the changes made in the first phase. In the final phase, we do the actual fix-point iteration by making as many nodes true (for least fix-points) as possible. When we have a set of changes to the transition system, we consider one change at a time i.e, execute the whole algorithm for an element in δ before considering any other element. Also, the lists used in the algorithm are sets i.e, there is no ordering between the elements and no multiple occurrences of any element in the list.

Algorithm

Phase 0: If a transition (s_m, s_n) is added to (deleted from) the LTS, then for all X_i and X_j such that $X_j = \forall \bigcirc X_i$ or $X_j = \exists \bigcirc X_i$ add(delete) the edge $\langle s_n, X_i \rangle \rightarrow \langle s_m, X_j \rangle$ to SV graph.

Phase 1(a):

- If $\langle s_n, X_i \rangle \rightarrow \langle s_m, X_j \rangle$ is added to SV graph and $X_j = \forall \bigcirc X_i$, and $\langle s_m, X_j \rangle$ is true and $\langle s_n, X_i \rangle$ is false then make $\langle s_m, X_j \rangle$ false.
- If $\langle s_n, X_i \rangle \rightarrow \langle s_m, X_j \rangle$ is added to SV graph and $X_j = \exists \bigcirc X_i$ and $\langle s_m, X_j \rangle$ is false and $\langle s_n, X_i \rangle$ is true then make $\langle s_m, X_j \rangle$ true.
- If $\langle s_n, X_i \rangle \rightarrow \langle s_m, X_j \rangle$ is deleted from SV graph and $X_j = \forall \bigcirc X_i$, and if $\langle s_n, X_i \rangle$ was its only false predecessor, then make $\langle s_m, X_j \rangle$ true.
- If $\langle s_n, X_i \rangle \rightarrow \langle s_m, X_j \rangle$ is deleted from SV graph and $X_j = \exists \bigcirc X_i$ and if $\langle s_n, X_i \rangle$ was its only true predecessor, then make $\langle s_m, X_j \rangle$ false.

Phase 1(b):

- for every edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ deleted such that both $\langle s, X_i \rangle$ and $\langle s', X_j \rangle$ are true, make $\langle s', X_j \rangle$ false and add it to F and also to another list F' .
- for every edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ added such that X_i and X_j come under the scope of the same fix-point operator, and both $\langle s, X_i \rangle$ and $\langle s', X_j \rangle$ are true, make $\langle s', X_j \rangle$ false and add it to F and also to another list F' .

Now we propagate these changes to the successors of the nodes whose values were updated in *phase 1(a)*. We define two kinds of lists of the SV graph nodes namely, the true list T consisting of all the nodes that were changed from false to true in *phase 1(a)* and the false list F defined in a dual manner.

Phase 2(a):

for each node $\langle s_m, X_i \rangle \in F$ do

- for each successor $\langle s_n, X_j \rangle$ of $\langle s_m, X_i \rangle$ such that $X_j = X_i \vee X_k$ or $X_j = \exists \bigcirc X_i$ do
 - if $\langle s_n, X_j \rangle$ is true and has no true predecessors then make $\langle s_n, X_j \rangle$ false and add $\langle s_n, X_j \rangle$ to F .
- for each successor $\langle s_n, X_j \rangle$ of $\langle s_m, X_i \rangle$ such that $X_j = X_i \wedge X_k$ or $X_j = \forall \bigcirc X_i$ do
 - if $\langle s_n, X_j \rangle$ is true, then make $\langle s_n, X_j \rangle$ false and add it to F .
- for each successor $\langle s_n, X_j \rangle$ of $\langle s_m, X_i \rangle$ such that $X_j = X_i \vee X_k$ or $X_j = \exists \bigcirc X_i$ do
 - if $\langle s_n, X_j \rangle$ is true and has a true predecessor then make $\langle s_n, X_j \rangle$ false and add $\langle s_n, X_j \rangle$ to F and F' .
- if $\langle s_n, X_j \rangle$ is the only successor of $\langle s_m, X_i \rangle$ and $X_j \in Var$ then
 - if $\langle s_n, X_j \rangle$ is true then make it false and add it to F .
- delete $\langle s_m, X_i \rangle$ from F .

Phase 2(b):

for each node $\langle s_m, X_i \rangle \in F'$ do

- if $(X_i = X_j \vee X_k \text{ or } X_i = \exists \bigcirc X_j)$ and $\langle s_m, X_i \rangle$ has a true predecessor then make $\langle s_m, X_i \rangle$ true and add it to T .
- if $(X_i = X_j \wedge X_k \text{ or } X_i = \forall \bigcirc X_j)$ and $\langle s_m, X_i \rangle$ has no false predecessors then make $\langle s_m, X_i \rangle$ true and add it to T .
- delete $\langle s_m, X_i \rangle$ from F' .

Phase 3:

for each node $\langle s_m, X_i \rangle \in T$ do

- for each successor $\langle s_n, X_j \rangle$ of $\langle s_m, X_i \rangle$ such that $X_j = X_i \wedge X_k$ or $X_j = \forall \bigcirc X_i$ do
 - if $\langle s_n, X_j \rangle$ is false and has no false predecessors then make $\langle s_n, X_j \rangle$ true and add $\langle s_n, X_j \rangle$ to T .

- for each successor $\langle s_n, X_j \rangle$ of $\langle s_m, X_i \rangle$ such that $X_j = X_i \vee X_k$ or $X_j = \exists \bigcirc X_i$ do
 - if $\langle s_n, X_j \rangle$ is false, then make $\langle s_n, X_j \rangle$ true and add it to T .
- if $\langle s_n, X_j \rangle$ is the only successor of $\langle s_m, X_i \rangle$ and $X_j \in Var$ then
 - if $\langle s_n, X_j \rangle$ is false then make it true and add it to T .
- delete $\langle s_m, X_i \rangle$ from T .

Note that if there is a nesting of fix-point operators in the formula, then the phases 2 and 3 have to be executed for each fix-point sub-formula in a bottom-up manner, i.e, the inner most fix-point sub-formula has to be evaluated first.

Let SV_1 be the final state-variable graph obtained after applying the above four steps. The set of states satisfying the fix-point formula in the new model are all states s such that $\langle s, X \rangle \in SV_1$ is true, where X is the variable appearing in the root of the evaluation tree T of the fix-point formula.

The above algorithm has a worst case complexity of the product of the sizes of the model and the formula, the size of model being the number of states and transitions and that of the formula being the number of equations in E . This is because, in the worst case we may have to visit each node of the SV graph. But in the best case, the complexity is linear in the size of the change applied to the model.

Consider the following simple example. The system consists of only 3 states with the following transition graph. The atomic proposition P is true only in state 3. Let the property to be verified be $\mu x.(P \vee \exists \bigcirc x)$ i. i.e, P will eventually be true along some path. The state-variable graph obtained is also given in the figure.

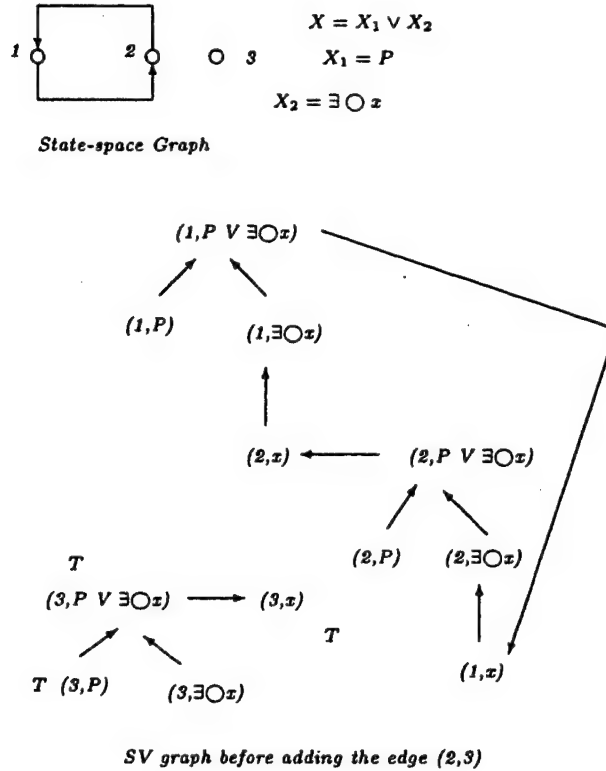


Figure 2. State Variable Graph Before Update.

Initially, assume all nodes are false. The bottom up evaluation results in the variable assignment which is represented in the figure. The node $\langle s, X \rangle$ labeled T implies X is true in state s . From the figure, we conclude that the fix-point formula is true only in state 3. Now, let a transition from state 2 to state 3 be added to the system. Using the incremental algorithm, this results in change in the variable assignment such that the formula is now satisfied in all three states of the system.

7.3 Comments

Consider the updated transition graph and its corresponding SV graph in the previous example. If the edge $(2,3)$ is deleted, we need to have a way for falsifying all the nodes in the cycle of the SV graph since the formula is no longer satisfied in states 1 and 2. This is done by the phase 1(b). That is, if we have a cycle or a strongly connected component in which all the nodes are of the same kind (i.e, either need all or atleast one of their predecessors to be true for their value to be true) then all the nodes in the cycle should be false for the variable assignment to be a fix-point solution.

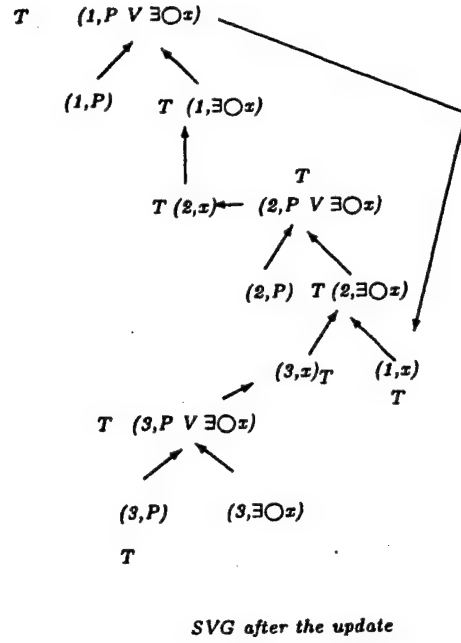


Figure 3. State Variable Graph After Update.

We now discuss the effect of different kinds of changes on the reachability graph.

- *An edge $s \rightarrow s'$ is added:* This results in addition of edges $\langle s', X_i \rangle \rightarrow \langle s, X_j \rangle$ for all relations of the form $X_j = \forall \bigcirc X_i$ or $X_j = \exists \bigcirc X_i$. When $X_j = \exists \bigcirc X_i$, $\langle s, X_j \rangle$ cannot become false if its already true and so the propagation of change stops at this node. But if $X_j = \forall \bigcirc X_i$, then $\langle s, X_j \rangle$ can become false and we need to propagate the changes further in the SV graph.
- *An edge $s \rightarrow s'$ is deleted:* This results in deletion of edges $\langle s', X_i \rangle \rightarrow \langle s, X_j \rangle$ for all relations of the form $X_j = \forall \bigcirc X_i$ or $X_j = \exists \bigcirc X_i$. When $X_j = \forall \bigcirc X_i$, $\langle s, X_j \rangle$ cannot become false if its already true and so the propagation of change stops at this node. But if $X_j = \exists \bigcirc X_i$, then $\langle s, X_j \rangle$ can become false and we need to propagate the changes further in the SV graph. This accounts for increase in cost of updating the SV graph.
- $L(S) \supset L'(S)$: The effect of decreasing the set of propositions that are true in state S is more costly than the case when $L(S) \subset L'(S)$. For example, if P is set to true in state 2 after adding the edge $(2, 3)$, it does not change values of any nodes except the node $\langle 2, P \rangle$. But if P is set to false in state 3, then the change has to be propagated further in the graph.

- *A new state S is added:* Here we have to add new nodes and edges and then use $L(S)$ to assign values to the new nodes and also propagate it to the already existing graph.
- *A state S is deleted:* First delete all the out going edges from all the nodes of the form $\langle S, X \rangle$ and compute its effect. Next delete these nodes and all the incoming edges to these nodes.

When there exists a cycle or strongly connected components in the SV graph such that all the nodes in them are of the same kind, then for the variable assignment, obtained after the update, to correspond to the least fix point, all the nodes in the cycle (or scc) should be false. For example, when the edge $(2, 3)$ is deleted we assume that the node $(2, \exists \bigcirc x)$ belongs to a cycle (containing all nodes of the same kind) and make it false even though one of its predecessors is still true, and propagate this change which makes all nodes of the cycle false. In cases when such an assumption is not correct, we undo the effect of such assumptions in the phase 2(b).

It is also important to note, from point of view of implementation, that when a user specifies a change to the system, the lower level representation of the system (which is the transition graph here) should also be updated incrementally before we can proceed to incrementally update the state-variable graph. In practice, it is very costly to represent the whole transition graph of the composition of the processes of the system. Hence, while constructing the global transition graph, only reachable states are considered. Also, every node in the SV graph has a counter associated with it that maintains the number of false (true) predecessors if the node requires all (some) of their predecessors to be true for it to be true.

8 Future Work

FRORL [1] is an architecture specification language that can express inheritance, non-monotonicity and object oriented features. It is to be extended to support architecture level features like interconnection relations between various objects, modularization, synchronous communication, event modeling. When the system consists of many processes executing independently, they have to be analyzed via compositional model-checking [11] and incremental approach to such compositional checking techniques. We are currently working on determining the kind of properties at the abstraction of architecture level that can be verified incrementally. We have also started the implementation phase and hope, by conducting several experiments, to analyze the efficiency of the overall incremental process from the higher level of user specification up to the construction and analysis of the state-

variable graph.

A lot of research has been done by people on representing real-time distributed systems using the architectural concepts. But very few have explored ways of integrating this with model checking i.e, integration of system description and its verification both at the level of architecture. We hope to improve upon the language given in this paper so that some good number of properties of such systems can be expressed and verified.

References

- [1]. J.J.-P. Tsai, T. Weigert, and H. Jang, A Hybrid Knowledge Representation as a Basis of Requirement Specification and Specification Analysis, *IEEE Trans. Software Engineering*, SE-18, No. 12, 1992.
- [2]. H. R. Andersen, Model Checking and Boolean graphs, *ESOP*, 1992.
- [3]. E. Allen Emerson, and Chin-Laung Lei, Efficient Model Checking in fragments of Propositional mu-calculus, *Logic in Computer Science*, 1986.
- [4]. R. Cleaveland, and B. Steffen, A linear time Model checking algorithm for Alternation free modal mu-calculus, *Formal Methods in System Design*, 1986.
- [5]. V. Sokolsky, and A. Smolka, Incremental model checking in modal mu-calculus, *6th International conference on Computer Aided verification*, 1994.
- [6]. E. A. Emerson, C. Jatla, and A. P. Sistla, Efficient Model-checking for fragments of mu-calculus, *International Conference on Computer Aided Verification*, Crete, Greece, June 1993.
- [7]. J. P. Tsai, and J Weigert, *Knowledge based software development for Real-time distributed systems*, World Scientific, 1994.
- [8]. D. Kozen, Results on Propositional mu-calculus, *Theoretical Computer Science*, 1983.
- [9]. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, Symbolic model checking: 10^{20} states and beyond. *IEEE Symp. Logic in Computer Science*, pp. 428-439, 1990.
- [10]. H. De-leon, and Orna Grumberg, Modular abstractions for verifying Real-time Distributed systems. *Formal methods in system design*, 1992.

[11]. E. M. Clarke, D. E. Long, and K. L. McMillan, Compositional Model checking, *4th Annual Symposium on LICS*, June, 1989.

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.